Network Security Final Report
Memory Bound Client Puzzles

Ruma Das
rdas@cs.jhu.edu

Sujata Doshi
sdoshi@cs.jhu.edu

December 8, 2004

1 Introduction

Client puzzles are computable cryptographic problems used to defend against connection depletion attacks [6]. These are constructed using time, secret information held by the server and additional client request information. When under attack, the server issues puzzles to the client requesting service. The client must solve the puzzle correctly and return the solution to the server in a certain time limit.

In Assignment 2 we evaluated CPU intensive client puzzles. We took a client's perspective and analyzed the number of puzzles that could be solved by the client in a given time bound, under varying sub-puzzle size (k). This led us to our interest in analyzing other forms of puzzles construction with regards to viability from clients perspective.

This design of the puzzle can be bound by the CPU constraints or the Memory constraints of the client. Memory bound puzzles overcome a significant obstacle in widespread adoption of client puzzles approach which occurs due to disparity in client CPU speed [1]. For example a high end system would solve a CPU-bound puzzle faster than a low end system, thereby making the CPU-bound puzzles unacceptable for wide deployment.

Memory access times vary much less compared to CPU speeds which makes Memory Bound puzzles an attractive choice for widespread deployment. In [1] Abadi *et. al.* validate the claim about memory access times, and propose a new memory bound construction. One of our aims is to validate their algorithm and compare it to the CPU bound variant [3]. We are interested in performing our experiments by taking a client side perspective.

An additional aim is to study the construction of memory bound puzzles using different data structures such as arrays, trees and registers. The practicality of the client puzzle approach depends not only on the processor speed, size of physical memory and cache but also on its effect on clients current state. Abadi *et. al.* pointed out that memory bound functions may interfere with currently running applications on the client machine and thus may not be acceptable to the users [1]. One of the goals of our project would be to analyze if it is possible to have memory bound puzzle construction that overcomes this problem.

In Section 2 we describe the primary related work in the area of Client Puzzles. In Section 3 we outline the goals of this project. Currently we have only addressed our first goal of Validation completely. We describe our validation experiments in Section 4. Based on primary related work we identify certain properties of memory bound puzzles which are indicated in Section 5.1. These properties will be useful when trying to come up with a different memory bound construction.

2 Related Work

Cryptographic puzzles were first introduced by Merkle [8]. Merkle applied these puzzles for key agreement protocols. In their protocol the session key lies in the solution the puzzle. For example if Alice and Bob wish to communicate, Bob can send Alice several puzzles which are moderately hard to solve. Alice will pick one puzzle at random and solve it and obtain the session key she can use for communication with Bob.

Juels *et. al.* further extended the idea of puzzles in [6]. Their focus was to come up with a counter-measure for connection depletion attacks. The client puzzle protocol overcomes some of the disadvantages of syn-cookies and the dropped connection approach. Their paper addresses most of the server side issues pertaining to this protocol. However they do not focus much on the clients perspective.

One of the approaches of building puzzles is the use of ”Hashcash” [3]. Hashcash is an example of CPU bound puzzles where interactive hashcash can be used for DOS throttling and graceful service degradation under CPU overload attacks on security protocols with computationally expensive connection establishment phases.

Hashcash differs from client puzzle suggest by Juels and Brainard in the sense that it is non interactive. This is because it was originally proposed as a counter-measure against email spam, where it is necessary to use non-interactive cost-functions as there is no channel for the server to send a challenge over.

Juels and Brainard’s [6] client-puzzle cost-function is an example of a interactive known-solution cost-function. The advantage of interactive cost-functions is that it is possible to prevent pre-computation attacks. In non-interactive cost functions, an attacker can spend a year pre-computing tokens to all be valid on the same day, and on that day be able to temporarily overload the system.

Another variation of client puzzles is the Time lock puzzle suggested by Rivest *et. al.* [9]. The goal of Time Lock puzzle is to encrypt a message so that it can not be decrypted by anyone not even the sender until a predetermined amount of time has passed. Time lock puzzles are computational problems that cannot be solved without running a computer continuously for at least a certain amount of time. The solution to the puzzle reveals a key that can be used to decrypt the encrypted information. Note that time lock puzzle require real time not total CPU time to solve. However Back points out [3] that the existence of a verification-key presents the added risk of key compromise allowing the attacker to by-pass the cost-function protection. In hashcash like client puzzle cost-function there is no key which would allow an attacker a short-cut in computing tokens. Another disadvantage of time-lock puzzle cost-function is that it will tend to have larger messages as there is a need to communicate planned and emergency re-keyed public parameters. Size of the puzzle is of importance as space is at a premium due to backwards compatibility and packet size constraints imposed by the network infrastructure

Dean *et. al.* describe a client puzzle extension to TLS [4]. Their results show that the client puzzle protocol is a feasible method for protecting SSL against denial of service attacks. They present a protocol design, which not only prevents against denial of service attacks, but is also backward compatible with the existing TLS implementation. In order to measure the effectiveness of their design, they modified the OpenSSL library to support their protocol specifications and benchmarked the system performance.

Wang and Reiter [11], try to handle the issues such as how to set the puzzle difficulty in the presence of an adversary with unknown computing power, and how to integrate the approach with existing mechanisms. There answer to this problem is a mechanism called as puzzle auction where each client bids for resources by tuning the difficulty of the puzzles it solves, and to adapt its bidding strategy in response to apparent attacks. Such kind of client puzzles auctions are effective option only when attackers have difficulty capturing vast computing resources. This many not be true due to available idle CPU cycles on internet. A workaround of this drawback is to replace the CPU bound puzzle construction with memory-bound puzzle construction for which the solution time is less variable as a function of the computing resources available to the attacker.

Waters *et. al.* [12], aim at using Client Puzzles to defend a server against DOS attacks. They point out that puzzle distribution itself can be subject to an attack. They have developed a new mechanism for distributing a puzzle using a robust service called *bastion*. They also propose a construction based on the Diffie Hellman algorithm where the bastion does not need to be aware of the servers using the system, and the puzzle solutions can be computed offline. They also point out that their new scheme can be used with several existing puzzle variants, including memory bound puzzles.

With respect to memory bound puzzles, Dwork *et. al.* consider memory bound constructions for fighting against spam mail [5]. The basic idea was to accompany email with certain computation effort, in order to reduce the motivation for sending unsolicited email. They propose an abstract function using random oracles and provide a lower bound on the amortized complexity of computing the proof of effort associated with the function. They also suggest implementations using the RC4 stream cipher. They performed rigorous tests on several machines. Their tests validate their claim that memory bound running times vary much less than CPU bound variants. Their tests also show that Pentium machines perform much better than MAC or SUN machines in the case of memory bound computations. They point out that their work has significant scope for extension; for example it would be desirable to have security proofs without resorting to the random oracle model.

In [10] the author examines the MBound algorithm presented by Dwork *et. al.* and derives the cost distribution for it. Based on their analysis they propose modified memory bound function for use in the LOCKSS protocol [7]. In their modified function, there is no uncertainity in the effort to be exerted by the client (generator of the proof). The server (verifier of the proof) can easily adjust the cost of the proof requested from the client. They still have to implement this modified algorithm in LOCKSS.

Abadi *et. al.* propose memory bound constructions in [1]. They believe the differences in CPU speed prevents wide spread deployment of the client puzzles which are CPU bound. They try to come up with new constructions for memory bound puzzles. Their constructions are originated from graph theory data structures. They also present a comparison of the Memory bound approach against the CPU bound approach [3].

In this project we explore further into Abadi *et. al.* paper and try to validate their results. We detail the basic algorithm presented by Abadi *et. al* in section 4. We identify the general properties of memory bound puzzles and based on these properties, we propose to come up with a memory bound construction. Our proposed goals are summarized in Section 3.

3 Proposed Goals and Approach

Our current goals are three fold.

1. **Validation:** We would first validate the Tree based experiment proposed in Abadi *et. al.*paper [1]. We would implement the tree based construction for client puzzles, proposed in this paper and evaluate it's performance from the client's perspective. An interesting analysis would be to determine how much does the depth of the tree, affect the number of puzzles the client can solve within a given time bound. For completeness we would also compare our results with CPU bound constructions [3].

2. **Memory Bound Constructions:** We would try to come up with different construction(s) for memory bound puzzles. We would try to obtain a construction which is time dependent and stateless as suggested in [6]. One of the approaches we plan to take is to look at computer architecture (for example

registers) based constructions of memory bound client puzzles. We also plan to determine what data structures (other than trees) could be used for constructing memory bound puzzles.

Abadi *et. al.* also suggest to look into models of memory hierarchy [2] and see if such models can be used in order to develop a foundation for memory bound computations. If time permits, we would also like to explore this line of research.

3. **Analysis:** We will evaluate the performance of these constructions and compare them with CPU bound constructions. Based on our results we would like to infer which type of puzzle constructions (CPU/Memory bound) are more suited for use in the real world from a clients perspective.

4 Validation

In this section we validate the experiments presented in Abadi *et. al.* [1]. We also compare our results to HashCash in Section 4.2.

4.1 Tree Based Experiment

Abadi *et. al.* [1] describe a puzzle in which the client is asked to reverse a one way function k times. Reversing the function by pure computation should take longer than a memory access, which motivates towards using memory bound techniques to solve the puzzle.

In other words building an answer table in memory and using it is preferable to actually computing the inverse function. The function may map some numbers of the domain to the same numbers in the range. This forces the client into exploring a tree to find the right answer. The server sends a checksum of the correct path when exploring the tree so that it need not spend much effort in validating a puzzle.

The basic memory bound algorithm described in the paper is as follows

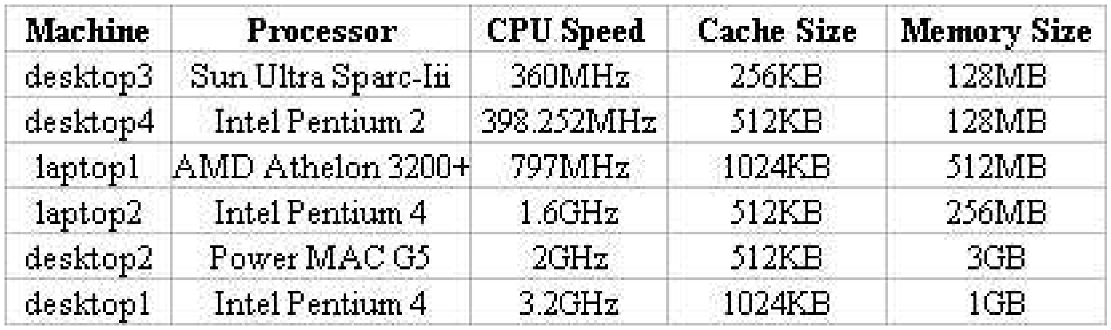
1. k and n are two integers known to both the client and the server. $F(\cdot)$ is a function whose domain and range are the integers $0 \dots 2^n - 1$. F is also known to both the client and the server.

2. The server starts of with an initial value x_0 and uses function $F(\cdot)$ to determine $x_{i+1} = F(x_i) \oplus i, i \epsilon \{0, \dots, k\}$. The server determines p such puzzles.

3. The server sends all the p puzzles, each consisting of x_k and a checksum over the path x_0 to x_k, to the client.

4. The client builds a table for $F^{-1}(\cdot)$. The same table is amortized over the p problems. The client performs p depth first searches for x_0. The client stops when he has found all correct p solutions. The correctness of the solution can be verified by the checksum.

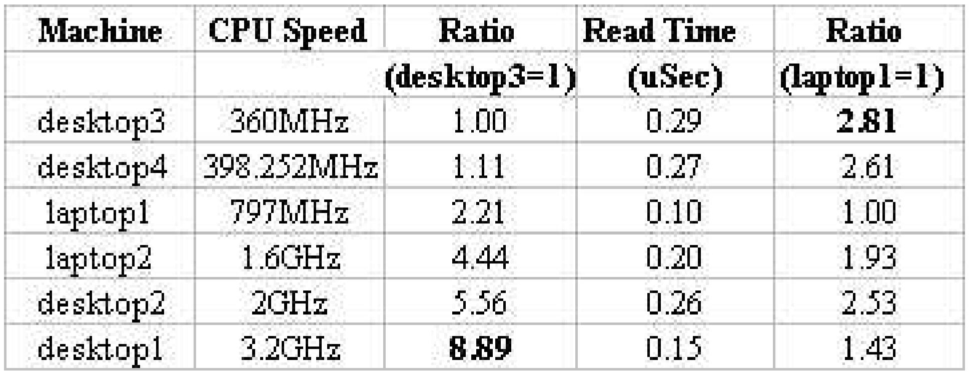
To ensure that the puzzle is indeed memory bound, the authors set $p = 128$, $k = 2^{11}$, $n = 22$. The authors also suggest that p could be set to 32 in which case the table build time would contribute no more than 25% of the total time to solve the p problems. In our validation experiments we have set $p = 32$. In the experiment, the function $F(\cdot) = middle\_bits(t_0[a_0] * t_1[a_1])$ where t_0, t_1 are two tables of $2^{n/2}$ random 32 bit numbers and a_0, a_1 are the bit strings obtained from x. The authors suggest the use of a cheap checksum for verification. In our implementation we used the $\sum_{i=0}^{k} x_i$ as the checksum; a better checksum would to use a hash function over the sequence $x_0 \dots x_k$.



Machine	Processor	CPU Speed	Cache Size	Memory Size
desktop3	Sun Ultra Sparc-Iii	360MHz	256KB	128MB
desktop4	Intel Pentium 2	398.252MHz	512KB	128MB
laptop1	AMD Athelon 3200+	797MHz	1024KB	512MB
laptop2	Intel Pentium 4	1.6GHz	512KB	256MB
desktop2	Power MAC G5	2GHz	512KB	3GB
desktop1	Intel Pentium 4	3.2GHz	1024KB	1GB

Figure 1: Machine Specifications

We implemented the basic algorithm in C++, and set the parameters as indicated above. We performed our tests on 5 different machines, whose specifications are given in Figure 1. In each of our experiments we have averaged over several sample points, however we have not removed any outliers because we did not see any such deviated points which would throw of our results.

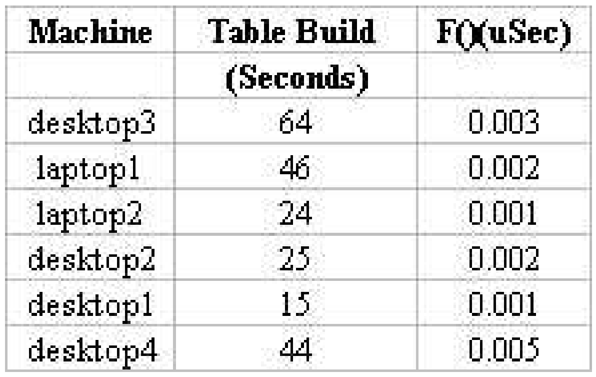


Machine	CPU Speed	Ratio	Read Time	Ratio
		(desktop3=1)	(uSec)	(laptop1=1)
desktop3	360MHz	1.00	0.29	**2.81**
desktop4	398.252MHz	1.11	0.27	2.61
laptop1	797MHz	2.21	0.10	1.00
laptop2	1.6GHz	4.44	0.20	1.93
desktop2	2GHz	5.56	0.26	2.53
desktop1	3.2GHz	**8.89**	0.15	1.43

Figure 2: Memory Bound Latencies on Different Machines

Figure 2 shows the Memory Read time measured on each of the machines. We performed million reads from random locations from an array of 2^{22} locations, each location being 8 bytes long. This is a table of $32MB$ which is greater than the maximum cache size. We measured the time taken to perform each read into the array, and also eliminated the overhead for performing the *gettimeofday* function calls. The authors on the other hand determined read time by measuring the time taken to follow a long linked list in memory. While our measurement methodology varies from that of Abadi *et. al.* our results are quite comparable to theirs. Now in Figure 2 we also compare the variation in read time to the variation in CPU speeds. It can be seen that while CPU speeds vary by a factor of 8.89, memory speeds vary only by a factor of 2.81. This validates Abadi *et. al.* claim that memory access times vary much less compared to CPU speeds.

Figure 3 shows the Table Build Time and the time taken for a single application of $F()$ on various machines. The table build time was averaged over 10 sample points. The function $F()$ on the other hand was applied a million times in order to capture the granularity of the time taken for a single application of $F()$. In the case of the Table Build Time, the latency is much higher compared to the latencies observed in the paper. We think this is due to the way we implemented our table. We used the standard template library map data structure in order to build our table. Each location in the table is indexed by $y = F(x)$ and the corresponding x values are stored in a $vector$ along side the key y. Now when building this table, in order to



Machine	Table Build (Seconds)	F()(uSec)
desktop3	64	0.003
laptop1	46	0.002
laptop2	24	0.001
desktop2	25	0.002
desktop1	15	0.001
desktop4	44	0.005

Figure 3: Table Build Time on Different Machines

resolve collisions, we are not just performing a single read, we are actually performing a $find$ function from the standard template library map data structure. In addition on a collision we update the $vector$ associated with the key y and to insert it back into the same key location, we first have to erase the old value and then write the new value. This leads to the overhead observed in our values.

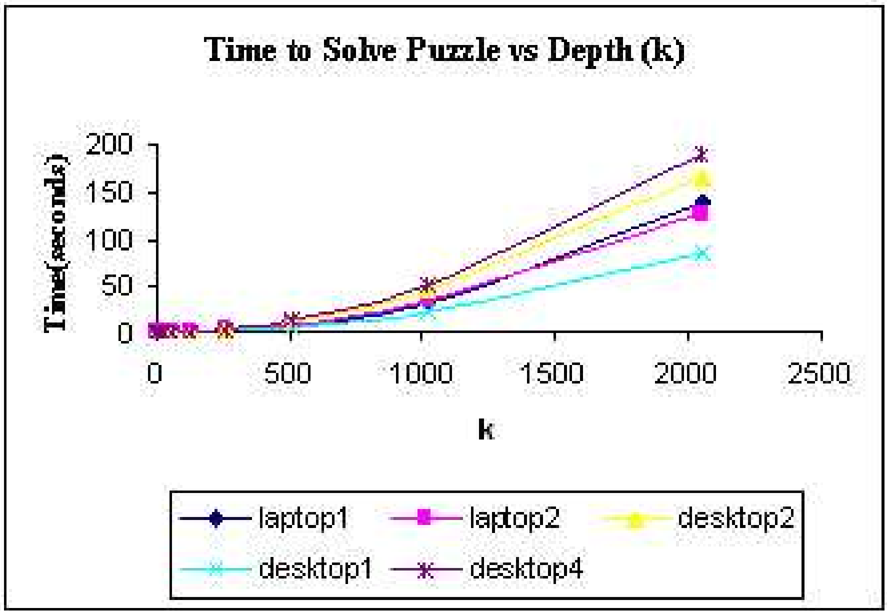


Figure 4: Time to Solve Puzzle against Tree Depth

Figure 4 shows the time taken by the client to solve the puzzle under varying depth k of the tree. We varied k from 2 to 2048 and measured the time for performing 32 depth first tree searches. The values are averaged over 10 runs of this experiment. As expected, the hardness of the puzzle increases with increasing k. It is interesting to see that the memory bound time does not vary so much across the different machines. The maximum variation is by a factor of 2.2.

In the next section we compare the memory bound algorithm to the CPU bound variant (HashCash).

4.2 Comparison to Hash Cash

In hash-cash, there is a CPU cost-function that computes a token which can be used as a proof-of-work. This cost function meets the properties of client puzzle that it is efficiently verifiable, but parametrically expensive to compute. Cost function in HashCash is called as Mint() which creates the token. The server will check the value of the token using an evaluation function VALUE(), and only proceed with the protocol or allocate resources if the token has the required value.

Minting of tokens essentially means finding a q bit partial collision in a hash function like SHA-1. In other words we have to find x and y, $x \neq y$ and the first q signifcant bits of $SHA-1(x)$ must match the same in $SHA-1(y)$. In the case of hashcash these first q bits should be zeros. Verification in HashCash is very simple, the server just has to use the VALUE() function to examine the first q bits, and verify that they are zero.

In [1] the authors compare their memory bound algorithm to hashcash. They evaluate the cost of minting 100 20 bit HashCash tokens. We also evaluated this mint time. We used the C implementation of HashCash in order to perform our experiments. Note that desktop3 (the SUN machines) is not present in our comparisons. This is because we were unable to get the C version of HashCash to work on SUN.

Figure 5 shows the time to mint the puzzle against the puzzle size. This time is averaged over 10 samples. It is interesting to see that across the various machines, the maximum variation in mint time is approximately a factor of 9 which is higher compared to our memory bound algorithm as depicted in Figure 4.

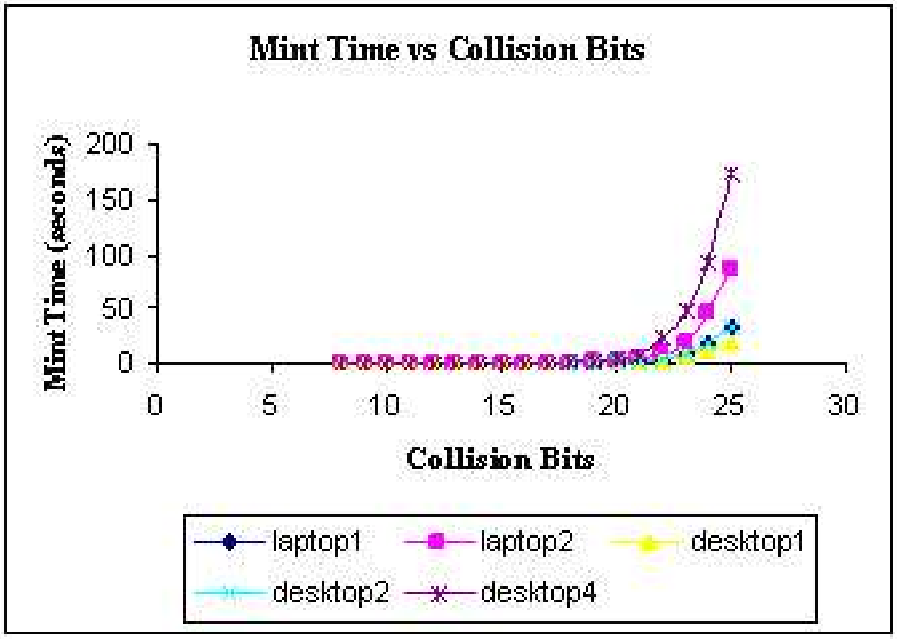
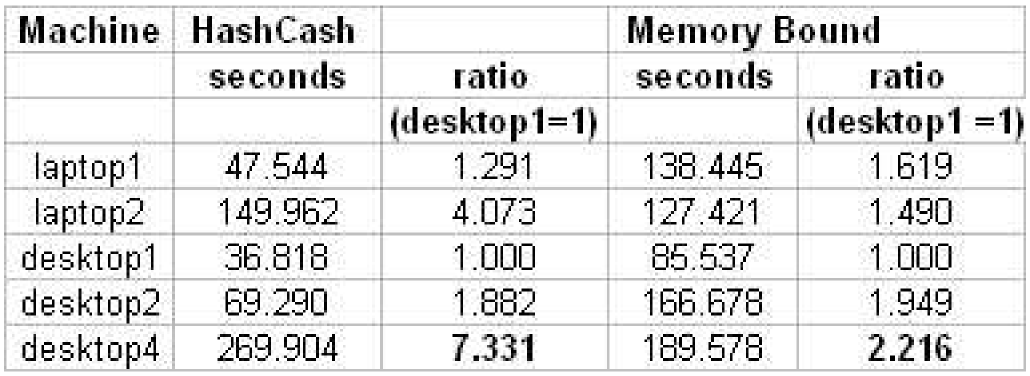


Figure 5: Time to Mint HashCash Puzzle against Bits

Figure 6 compares the results for minting 100 20 bit puzzles against solving the memory bound puzzle ($n = 22, k = 2048, p = 32$). The mint time taken on various machines was much faster compared to the results presented by the author. We are not sure why this is the case; it is possible that the authors used a different implementation of HashCash for testing .

The memory bound times exclude the table building time. The memory bound numbers are higher than those determined by the author. This is possibly due to the way we implemented our inverse table for $F()$. The main point to notice from this table is that the memory bound ratio across various machines remains under 2.2 whereas CPU bound ratio is much higher and lies under 7.3.



Machine	HashCash		Memory Bound	
	seconds	ratio	seconds	ratio
		(desktop1=1)		(desktop1 =1)
laptop1	47.544	1.291	138.445	1.619
laptop2	149.962	4.073	127.421	1.490
desktop1	36.818	1.000	85.537	1.000
desktop2	69.290	1.882	166.678	1.949
desktop4	269.904	**7.331**	189.578	**2.216**

Figure 6: Comparison of Hashcash and Memory Bound Algorithm

While our validation experiments do not yield similar numbers as that of the authors, they still bring out the Abadi *et. al.* point that memory bound times do not vary much across different machines, as compared to CPU bound times. Further the memory bound function performs better on Intel machines (desktop1 and laptop2) compared to MAC machines. This result is also brought up in [5]. They point out that the MAC and SUN machines do not perform as well in memory bound computations due to its poor handling of Translation Lookahead Buffer misses.

In the next section we present certain properties of client puzzles which can be useful when trying to come up with a new construction.

5 Memory Bound Constructions

In this section we first enumerate the general properties of client puzzles , and also properties specific to memory bound puzzles.

5.1 Properties of Client Puzzles

Properties of client puzzles have been presented in many of the related works. We have tried to compile these properties into the following definition.

Definition 1 ***Client Puzzles*** *are computable cryptographic problems which are in accordance with the following properties*

- ***Stateless:*** *The server should be able to verify the puzzle solution without maintaining any database.*
- ***Time-Dependant:*** *The client should be given limited time to solve the puzzle.*
- ***In-expensive Server-Side Cost:*** *Creation and verification of the puzzle should be inexpensive to the server.*
- ***Controlled Hardness:*** *The server should be able to control the hardness of the puzzle sent to the client.*

- ***Hardware Independant:*** *The puzzle should not be hardware dependant. This property ensures that the puzzle can be widely deployed.*
- ***Impossibility of Pre-computation:****The puzzle solution cannot be pre-computed by the client. This ensures that while the puzzle can be reused, its solution is not reusable.*

Memory bound client puzzles should not only follow the above properties, but should also be constructed in accordance to the following properties.

- ***Random Memory Access:*** *A memory bound function should access random memory locations in such a way that the cache memory becomes ineffective.*
- ***Slower CPU bound solution variants:*** *A client puzzle, can be solved by a memory bound method or a CPU bound method. In order to make the puzzle memory bound, the Memory Bound algorithm should converge faster than the corresponding CPU bound variant.*

The above properties are generic and will be useful when trying to come up with a different memory bound construction.

6 Conclusion and Future Work

The goal of the project is to study the memory bound puzzles and evaluate its applicability as compared to CPU bound puzzles. We were able to verify that across different machines, CPU speeds vary by a larger factor as compared to memory latency. Comparing the Tree based memory bound experiment with HashCash based CPU bound experiment, we were able to show that memory bound puzzle cost varied by a small factor as compared to CPU bound puzzle.

Another point brought out by the tree based experiment is that we can vary the value of the parameters n, k, p etc to evolve with the changing enhancement in the memory technology. For e.g. if the cache sizes increases, the parameter n can be increased so that inverse function table will still be larger than cache but fit into the memory. This leads to infer that memory bound puzzle constructions are more suited for use in the real world from a client's perspective.

However we are aware of the issues that prevent the memory bound constructions form being accepted. Abadi et. al. mention that memory bound functions can interfere with concurrently running applications in a multitasking environment. In order to solve this problem we would need to come up with a puzzle construction that would use instructions that can access memory without going through the cache. Due to time limitations we were not able to explore such possibilities. Exploring possible new constructions for memory bound puzzles would be an important topic which we would like to address as our future work.

References

[1] Martin Abadi, Mike Burrows, Mark Manasse, and Ted Wobber. Moderately hard, memory-bound functions.

[2] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 305–314, 1987.

[3] A. Back. Hash cash - a denial of service counter-measure.

[4] D. Dean and A. Stubblefield. Using client puzzles to protect tls. In *Proceedings of the 10th USENIX UNIX Security Symposium*, 2001.

[5] C. Dwork, A. Goldberg, and M. Naor. On memory-bound functions for fighting spam, 2002.

[6] A. Juels and J. Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks, 1999.

[7] P. Maniatis, M. Roussopoulos, T. Giuli, D. Rosenthal, M. Baker, and Y. Muliadi. Preserving peer replicas by rate-limited sampled voting, 2003.

[8] R. C. Merkle. Secure communications over insecure channels. *Communications of ACM*, 21(4):294–299, 1978.

[9] R. L. Rivest, A. Shamir, and D. A. Wagner. Time-lock puzzles and timed-release crypto. Technical Report MIT/LCS/TR-684, 1996.

[10] David Rosenthal. On the cost distribution of a memory bound function.

[11] XiaoFeng Wang and Michael K. Reiter. Defending against denial-of-service attacks with puzzle auctions. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 78. IEEE Computer Society, 2003.

[12] Brent Waters, Ari Juels, J. Alex Halderman, and Edward W. Felten. New client puzzle outsourcing techniques for dos resistance.